Pipelining

What is pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs.

Throughput of an instruction pipeline is determined by how often an instruction exits the pipeline.

Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time. The time required between moving an instruction one step down the pipeline is a *processor cycle*. Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage.

Pipelining yields a reduction in the average execution time per instruction. Depending on what you consider as the baseline, the reduction can be viewed as decreasing the number of clock cycles per instruction (CPI), as decreasing the clock cycle time, or as a combination of the two things.

A simple implementation of a RISC instruction set

To understand how a RISC (Reduced Instruction Set Computer) instruction set can be implemented in a pipelined fashion, we need to understand how it is implemented *without* pipelining. Here, we show a simple implementation where every instruction takes at most five clock cycles.

The five clock cycles are as follows:

Name of stage	Acronym	Description
Instruction fetch cycle	IF	Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 to it.
Instruction decode/register fetch cycle	ID	Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed. Compute the

Name of stage	Acronym	Description
		possible brach target address by adding the sign extended offset to the incremented PC.
Execution/effective address cycle	EX	The ALU (Arithmetic Logic Unit) operates on the operands prepared in the prior cycle, performing one of three functions, depending on the instruction type: (i) memory reference: the ALU adds the base register and the offset to form the effective address; (ii) register-register ALU instruction: the ALU performs the operation specified by the ALU opcode on the values read from the register file; (iii) register-immediate ALU instruction: the ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.[1]
Memory access	MEM	If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.
Write-back cycle	WB	For register-register ALU instructions and load instructions, write the result into the register file, whether it comes from the memory or from the ALU.

In this implementation, branch instructions require two cycles, store instructions require four and all other instructions require five. Assuming a branch frequency of 12% and a store frequency of 10%, a typical instruction distribution leads to an overall CPI of 4.54.

The classic five-stage pipeline for a RISC processor

We can pipeline the execution described above with almost no changes by simply starting a new instruction on each clock cycle. Each of the clock cycles from the previous section becomes a *pipe stage*—a cycle in the pipeline. This results in the execution pattern shown below:

Instruction\Clock	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Unfortunately, pipelining is not that simple: first of all, we must determine what happens on every clock cycle of the processor and make sure we don’t try to perform two different operations with the same data path resource on the same clock cycle. Thus, we must ensure that the overlap of instructions in the pipeline cannot cause such a conflict. Fortunately, the simplicity of a RISC instruction set makes resource evaluation relatively easy: the major functional units are used in different cycles of the pipeline, and hence overlapping the execution of multiple instruction introduces relatively few conflicts. There are three observations on which this fact rests:

1. We use separate instruction and data memories, which we would typically implement with separate instruction and data caches. This eliminates a conflict for a single memory which would arise between instruction fetch and data memory access.
2. The register file is used in the two stages: one for reading in ID and one for writing in WB. These uses are distinct, so we need to perform two reads and one write every clock cycle. To handle reads and a write to the same register, we perform the register write in the first half of the clock cycle and the read in the second half[2]
3. To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage, in preparation for the next instruction. Furthermore, we must also have an adder to compute the potential branch target address during ID.

We must also ensure that the instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing *pipeline registers* between successive stages of the pipeline, so that at the end of a clock cycle, all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle. It is sometimes useful to name the pipeline registers, and we follow the convention of naming them by the pipeline stages they connect: IF/ID, ID/EX, EX/MEM and MEM/WB.

Basic performance issues in pipelining

Pipelining increases the CPU instruction throughput—the number of instructions completed per unit of time—but it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction, due to overhead in the control of the pipeline. The fact that the execution time of each instruction does not decrease puts limits on the practical depth of a pipeline.

In addition to limitations arising from pipeline latency, limits arise from imbalance among the pipe stages and from pipelining overhead.

Pipeline hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.

There are three classes of hazards:

1. **Structural** hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution
2. **Data** hazards arise when an instruction depends on the results of a previous instruction
3. **Control** hazards arise form the pipelining of branches and other instructions that change the PC

Hazards in pipelines can make it necessary to *stall* the pipeline. When an instruction is stalled, all instructions issued later than the stalled instruction are also stalled, while instructions issued earlier than the stalled instruction must continue. No new instructions are fetched during the stall.

Structural hazards

When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a *structural hazard.*

The most common instances of structural hazards arise when some functional unit is not fully pipelined. Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

When a sequence of instructions encounters this hazard, the pipeline will stall one of the instructions until the required unit is available. Such stalls will increase the CPI from its usual ideal value of 1.

If all factors are equal, a processor without structural hazards will *always* have a lower CPI. Sometimes, designers will allow structural hazards in order to lower the cost of the unit, since duplicating functional units can be costly.

Data hazards

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.

Consider the pipelined execution of these instructions:

```
DADD    R1, R2, R3
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

All the instructions after the `DADD` use the result of the `DADD` instruction. The `DADD` instruction writes the value of R1 in the WB pipe stage, but the `DSUB` instruction reads the value during its ID stage. This problem is called *data hazard*.

Unless precautions are taken to prevent it, the `DSUB` instruction will read the wrong value and try to use it. This unpredictable behaviour is obviously unacceptable.

The `AND` instruction is also affected by this hazard: the write of R1 does not complete until the end of clock cycle 5. Thus, the `AND` instruction that reads the registers during clock cycle 4 will receive the wrong results.

The `XOR` instruction operates properly because its register read occurs in clock cycle 6, after the register write. The `OR` instruction also operates without incurring a hazard, because we perform the register file reads in the second half of the cycle and the writes in the first half.

Minimising data hazard stalls by forwarding

The problem posed above can be solved with a simple hardware technique called *forwarding*. The key insight in forwarding is that the result is not really needed by the `DSUB` until after the `DADD` instruction actually produces it. If the result can be moved from the pipeline register where the `DADD` stores it to where the `DSUB` needs it, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control

logic selects the forwarded result as the ALU input rather than the value read from the register file.

Forwarding can be generalised to include passing a result directly to the functional unit that requires it: a result is forwarded from the pipeline register corresponding to the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit. Take, for example, the following sequence:

```
DADD    R1, R2, R3
LD      R4, 0(R1)
SD      R4, 12(R1)
```

To prevent a stall in this sequence, we would need to forward the values of the ALU output and memory unit output from the pipeline registers to the ALU and data memory inputs.

Unfortunately, not all potential data hazards can be handled by forwarding.

Control hazards

Control hazards can cause a greater performance loss for our MIPS pipeline than do data hazards.

When a branch is executed, it may or may not change the PC to something other than its current value plus 4.

Recall that if a branch changes the PC to its target address, it is a taken branch; if it falls through, it is not taken.

If instruction i is a taken branch, then the PC is normally not changed until the end of ID, after the completion of the address calculation and comparison.

The simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID. This means that the first IF cycle is essentially a stall, because it never performs useful work.

If the branch is untaken, the repetition of the IF stage is unnecessary, since the correct instruction was indeed fetched.

One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency.

---

1. In a load-store architecture, the effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to simultaneously calculate a data address and perform an operation on the data.↩
2. Exploiting the rising- and falling-edge of the clock.↩